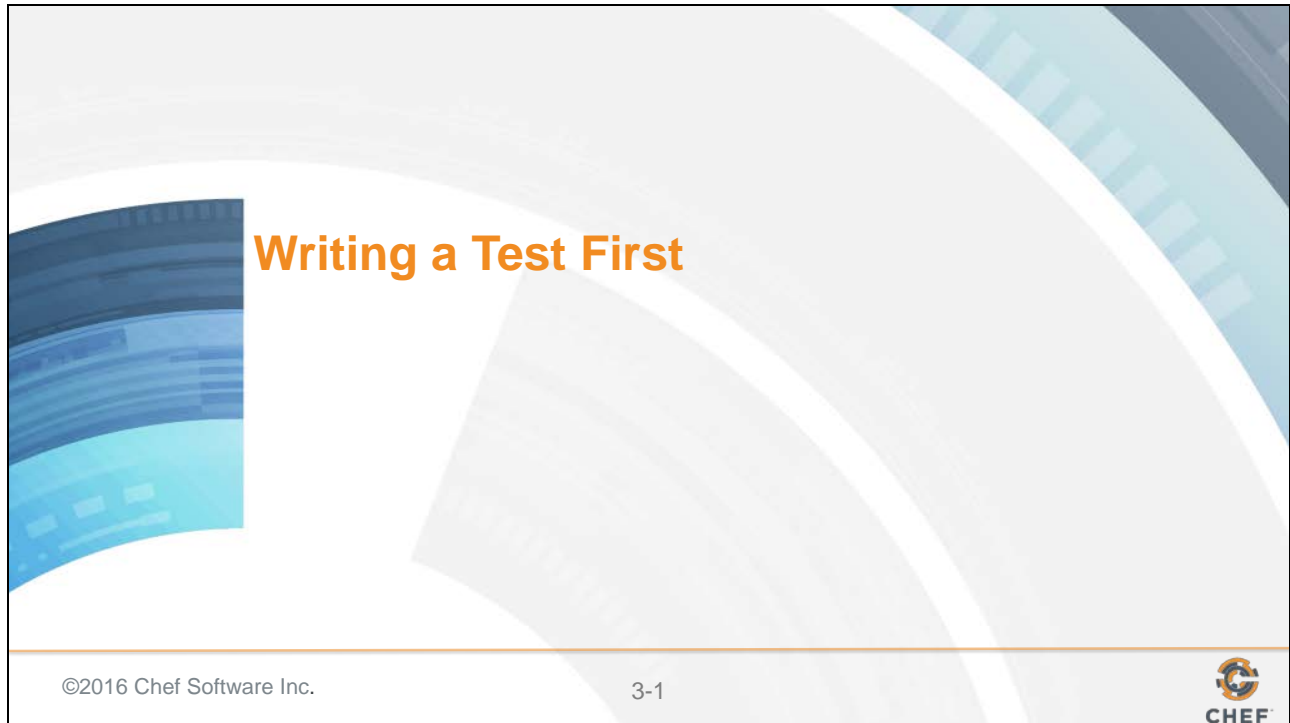


### 3: Writing a Test First




Writing tests are often difficult. Writing tests before you have written the code that you want to test can often feel like a leap of faith. An act that requires a level of clairvoyance reserved for magicians or con-artists. Some have likened it towards starting a story by first writing the conclusion.

## Slide 2

# CONCEPT


## Test Driven Development

1. Define a test set for the unit first
2. Then implement the unit
3. Finally verify that the implementation of the unit makes the tests succeed.
4. Refactor



©2016 Chef Software Inc.

3-2




Test Driven Development (TDD) is a workflow that asks you to perform that act continually and repeatedly as you satisfy the requirements of the work you have chosen to perform.

TDD generically focuses on the unit of software any level. It is the process of writing the test first, implementing the unit, and then verifying the implementation with the test that was written.

A 'unit' of software is purposefully vague. This 'unit' is definable by the individuals developing the software. So the size of a 'unit of software' likely has different meanings to different individuals based on our backgrounds and experiences.

## Slide 3

# CONCEPT




## Behavior Driven Development (BDD)

Behavior-driven development (BDD) specifies that tests of any unit of software should be specified in terms of the desired behavior of the unit.

Borrowing from [agile software development](#) the "desired behavior" in this case consists of the requirements set by the business — that is, the desired behavior that has [business value](#) for whatever entity commissioned the software unit under construction.

Within BDD practice, this is referred to as BDD being an "outside-in" activity.

---


©2016 Chef Software Inc. 3-3 

How you choose to express the requirements of that unit is the crux of Behavior Driven Development (BDD). Behavior-driven development specifies that tests of any unit of software should be specified in terms of the desired behavior of the unit. Expressing this desired behavior is often expressed in scenarios that are written in a Domain Specific Language (DSL).

The cookbooks and recipes that you have written so far share quite a few similarities with BDD. In Chef, you express the desired state of the system through a DSL, resources, you define in recipes.

## Slide 4

# CONCEPT




## TDD and BDD

**TDD** is a workflow process.

**BDD** influences the language we use to write tests and how we focus on the tests that matter.

©2016 Chef Software Inc.

3-4



TDD is a workflow process: Add a test; Run the test expecting failure; Add code; Run the test expecting success. Refactor.

BDD influences the language we use to write the tests and how we focus on tests that matter. The activities within this module focus on the process of taking requirements, expressing them as expectations, choosing one implementation to meet these expectations, and then verifying we have met these expectations.

## Slide 5

## Objectives

After completing this module, you should be able to:

- Use chef to generate a cookbook
- Write an integration test
- Use Test Kitchen to create, converge, and verify a recipe
- Develop a cookbook with a test-driven approach

In this module you will learn how to use chef to generate a cookbook, write an integration test first, use Test Kitchen to execute that test, and then implement a solution to make that test pass.

## Slide 6

## Building a Web Server

1. Install the httpd package
2. Write out a test page
3. Start and enable the httpd service

To explore the concepts of Test Driven Development through Behavior Driven Design we are going to focus on creating a cookbook that starts with the goal that installs, configures, and starts a web server that hosts the your company's future home page.

This cookbook will start very straight-forward and over the course of these modules we will introduce new requirements that will increase its complexity.

The goal again is to focus on the TDD workflow and understanding how to apply BDD when defining these tests. We are not concerned about focusing on best practices for managing web servers or modeling a more initially complex cookbook.

## Slide 7

## Defining Scenarios

**Given** SOME CONDITIONS

**When an** EVENT OCCURS

**Then I should** EXPECT THIS RESULT

When requirements come to us it is rare that the product owners and customers ask us to deliver a particular technology or a software. In our case, I have asked you to setup a web page for your company. I did not specifically state a particular technology but to help limit the scope I have chosen that we are going to build this initial website with Apache.

Behavior driven design asks us to look at the work that we perform from the perspective of our users. Our first job is to develop the scenario that validates the work that we are about to accomplish.

These scenarios that we write are often written in the following format.

This very generically defines any scenario. What we need to do is apply this scenario format to our requirements.

## Slide 8

## The Why Stack?

You should discuss...the feature and [pop the why stack](#) max 5 times (ask why recursively) until you end up with one of the following business values:

- Protect revenue
- Increase revenue
- Manage cost

If you're about to implement a feature that doesn't support one of those values, chances are you're about to implement a non-valuable feature. Consider tossing it altogether or pushing it down in your backlog.

- Aslak Hellesøy, creator of Cucumber

If our goal is to setup a new webpage we need to start to ask ourselves the question: Why. Why do we need to setup a website? Asking this question will help us identify for who the website is for and what purpose does it serve for the actor in this scenario.

Often times the why will raise more questions which you continue to ask why. You should do that. Asking why enough times will lead you to the true reason why you are taking action. The interesting thing is that knowing the true reason why will help reinforce your course of action or maybe change it entirely.



## Slide 9

## Scenario: Potential User Visits Website

Given that **I am a potential user**

When **I visit the company website in my browser**

Then I should **see a welcome message**

The typical reason for setting up a website is to allow customers, users, potential users to learn more about the company. The needs of the website may change in the future but the first minimum viable product (MVP) is to simply give our users the ability to find out more information.


Our goal now is to define a scenario with this understanding.

This first scenario is enough information to help us build this cookbook with a TDD approach. This practice of defining a scenario is a tactic that I employ to help focus me on the most valuable work that needs to be done.

Important things to notice in the following scenario is the distinct lack of technology or implementation. The scenario is not concerned about the services that are running or files that might be found on the file system.

## Slide 10

# EXERCISE




## Build a Reliable Cookbook

*This time it will be different.*

**Objective:**

- ☐ Create a cookbook
- ☐ Write tests that verifies the cookbook does what we want it to do
- ☐ Execute the tests and see failure
- ☐ Write the recipe to make the test pass
- ☐ Execute the tests and see success


---

©2016 Chef Software Inc. 3-10 

With the scenario defined it is now time for us to develop the cookbook. We are going to move through the following steps together to accomplish this task.

## Slide 11

## Let's Start this Journey in the Home Directory



A terminal window icon is shown on the left. The terminal window has a black background with white text. The prompt is `>` and the command entered is `cd ~`. The rest of the terminal window is empty.

©2016 Chef Software Inc. 3-11



Let's start the journey on your workstation. From the home directory we are going to creating this cookbook.

## Ask Chef About Generating a Cookbook



```
> chef generate --help
```

```
Usage: chef generate GENERATOR [options]
```

```
Available generators:
```

app	Generate an application repo
cookbook	Generate a single cookbook
recipe	Generate a new recipe
attribute	Generate an attributes file
template	Generate a file template
file	Generate a cookbook file
lwrp	Generate a lightweight resource/provider
repo	Generate a Chef code repository
policyfile	Generate a Policyfile for use with the install/push commands

There are a number of tools installed with the Chef Development Kit (Chef DK). One of those tools included in the Chef DK is a tool called 'chef'. The generators provided with the tool will allow us to quickly generate the a cookbook. You can see help about the command with the '--help' flag.

The cookbook generator has only one required parameter and that is the name of the cookbook.

## Generate a Cookbook



```
> chef generate cookbook httpd
```

```
Compiling Cookbooks...
```

```
Recipe: code_generator::cookbook
```

```
  * directory[/home/chef/httpd] action create
```

```
    - create new directory /home/chef/httpd
```

```
  * template[/home/chef/httpd/metadata.rb] action  
create_if_missing
```

```
    - create new file /home/chef/httpd/metadata.rb
```

```
    - update content in file /home/chef/httpd/metadata.rb from  
none to 53a150
```

```
    (diff output suppressed by config)
```

```
  * template[/home/chef/httpd/README.md] action create_if_missing
```

Let's generate cookbook named 'httpd'. The name of the cookbook here resembles the name of a public cookbook in the Supermarket that accomplishes a very similar task. That is the reason why I have asked you to choose the same name.

Sharing the same name as a cookbook within the Supermarket can be problematic. While we may never share this cookbook other individuals within our organization could believe it to be a copy of that cookbook. When it comes to naming cookbooks it may be wise to first search the Supermarket and ensure you are not using a similar name.

## View the Tests in the Generated Cookbook




```
> tree httpd
```

```
httpd
├── test
│   ├── integration
│   │   ├── default
│   │   │   └── serverspec
│   │   │       └── default_spec.rb
│   └── helpers
│       ├── serverspec
│       └── spec_helper.rb
```

We can examine the contents of the cookbook that chef generated for us. Here you see that the tool created for us a complete test directory structure.

## Slide 15

# EXERCISE




## Build a Reliable Cookbook

*This time it will be different.*

**Objective:**

- ✓ Create a cookbook
- ☐ Write tests that verifies the cookbook does what we want it to do
- ☐ Execute the tests and see failure
- ☐ Write the recipe to make the test pass
- ☐ Execute the tests and see success

---

©2016 Chef Software Inc. 3-15 

With the cookbook created it is now time to write that first test that verifies the cookbook does what we want it to do.

## Slide 16

# CONCEPT

## RSpec and ServerSpec

RSpec is a Domain Specific Language (DSL) that allows you to express and execute expectations. These expectations are expressed in examples that are asserted in different example groups.

ServerSpec provides helpers and tools that allow you to express expectations about the state of infrastructure.

ServerSpec  
RSpec Chef  
Ruby

©2016 Chef Software Inc.

3-16

RSpec is a Behavior Driven Development (BDD) framework that uses a natural language domain-specific language (DSL) to quickly describe scenarios in which systems are being tested. RSpec allows you to setup a scenario, execute the scenario, and then define expectations on the results. These expectations are expressed in examples that are asserted in different example groups.

RSpec by itself grants us the framework, language, and tools. ServerSpec provides the knowledge about expressing expectations about the state of infrastructure.



## Auto-generated Spec File in Cookbook

 ~/httpd/test/integration/default/serverspec/default\_spec.rb


```
require 'spec_helper'

describe 'httpd::default' do
  # Serverspec examples can be found at
  # http://serverspec.org/resource_types.html
  it 'does something' do
    skip 'Replace this with meaningful tests'
  end
end
```

The generator created an example specification (or spec) file. Before we talk about the RSpec/ServerSpec language lets explain the long file path and its importance.

## Slide 18

# CONCEPT



## Where do Tests Live?

```
~/httpd/test/integration/default/serverspec/default_spec.rb
```


Test Kitchen will look for tests to run under this directory. It allows you to put unit or other tests in test/unit, spec, acceptance, or wherever without mixing them up. This is configurable, if desired.

<http://kitchen.ci/docs/getting-started/writing-test>

---

©2016 Chef Software Inc.


3-18



Let's take a moment to describe the reason behind this long directory path. Within our cookbook we define a test directory and within that test directory we define another directory named 'integration'. This is the basic file path that Test Kitchen expects to find the specifications that we have defined.

## Slide 19

# CONCEPT



## Where do Tests Live?

```
~/httpd/test/integration/default/serverspec/default_spec.rb
```


This corresponds exactly to the Suite name we set up in the .kitchen.yml file. If we had a suite called "server-only", then you would put tests for the server only suite under.

<http://kitchen.ci/docs/getting-started/writing-test>

---

©2016 Chef Software Inc.


3-19



The next part the path, 'default', corresponds to the name of the test suite that is defined in the .kitchen.yml file. In our case the name of the suite is 'default' so when test kitchen performs a `kitchen verify` for the default suite it will look within the 'default' folder for the specifications to run.

## Slide 20

# CONCEPT



## Where do Tests Live?

```
~/httpd/test/integration/default/serverspec/default_spec.rb
```


This tells Test Kitchen (and Busser) which Busser runner plugin needs to be installed on the remote instance.

<http://kitchen.ci/docs/getting-started/writing-test>

---

©2016 Chef Software Inc.


3-20



'serverspec' is the kind of tests that we want to define. Test Kitchen supports a number of testing frameworks.

## Slide 21

# CONCEPT



## Where do Tests Live?

```
~/httpd/test/integration/default/serverspec/default_spec.rb
```


All test files (or specs) are named after the recipe they test and end with the suffix "\_spec.rb". A spec missing that will not be found when executing `kitchen verify`.

<http://kitchen.ci/docs/getting-started/writing-test>

---

©2016 Chef Software Inc.

3-21



'serverspec' is the kind of tests that we want to define. Test Kitchen supports a number of testing frameworks.

Now that we understand the path let's take a look at the RSpec/ServerSpec language.

## Slide 22

## ServerSpec Example

```
describe 'httpd::default' do
  describe command('curl http://localhost') do
    its(:stdout) { should match(/Welcome Home/) }
  end
end
```

example groups

ServerSpec allows you to scope the expectations you right into groups. These groups are given names or ServerSpec resources to help describe their scope. Example groups can be nested and often are to describe a hierarchical structure of state.

In this example we see an outer example group with the name 'httpd::default' and the inner example group with the ServerSpec resource named command. This means that the command example group is within the example group 'httpd::default'.

Example groups can be used to show relationship, like they are done here. They can also be used to create unique scenarios with different states for the system.

## Slide 23

## Components of a ServerSpec Example

```
describe 'httpd::default' do
  describe command('curl http://localhost') do
    its(:stdout) { should match(/Welcome Home/) }
  end
end
```

The diagram illustrates the components of the provided ServerSpec example. Callouts point to specific parts of the code:

- cookbook name::suite name**: Points to the outermost `describe 'httpd::default' do` block.
- ServerSpec resource**: Points to the `describe command('curl http://localhost')` block.
- attribute of resource**: Points to the `its(:stdout)` attribute query.
- expectation**: Points to the `should match(/Welcome Home/)` expectation.

[http://serverspec.org/resource\\_types.html](http://serverspec.org/resource_types.html)

The outermost describe here represents the cookbook name and the suite name. The inner expectation is using a ServerSpec resource named command. This command resource takes a parameter which is the system command it will run. Within the inner example group we are asking for an attribute on the ServerSpec command resource.

A command resource is one of the few resources that has attributes you can query. In this instance the example is asking for the standard out, abbreviated as 'stdout'. Lastly we express an expectation that we expect the actual result returned from the standard out to contain the text 'Welcome Home'.

## Slide 24

## Remove the Default Test

```
~/httpd/test/integration/default/serverspec/default_spec.rb
```

```
require 'spec_helper'

describe 'httpd::default' do
  # Serverspec examples can be found at
  # http://serverspec.org/resource_types.html
  it 'does something' do
    skip 'Replace this with meaningful tests'
  end
end
```

Within the test file found at the following path you will find that it is already populated with some initial code. The first two lines are comments that provide a link to the ServerSpec documentation. The next three lines are a placeholder test that when executed notifies you that this test is skipped.

We do not need these comments or the placeholder test so let's remove it from the specification.



## Add a Test to Validate a Working Website

~/httpd/test/integration/default/serverspec/default\_spec.rb

```
require 'spec_helper'


describe 'httpd::default' do
  describe command('curl http://localhost') do
    its(:stdout) { should match(/Welcome Home/) }
  end
end
```

Define the following expectation that states that when we would visit that site it should return a welcoming message.

ServerSpec provides a helper method that allows you to specify a command. That command returns the results from the command through standard out. We are asking the command's standard out if anywhere in the results match the value 'Welcome Home'.

## Slide 26

# EXERCISE




## Build a Reliable Cookbook

*This time it will be different.*

**Objective:**

- ✓ Create a cookbook
- ✓ Write tests that verifies the cookbook does what we want it to do
- ☐ Execute the tests and see failure
- ☐ Write the recipe to make the test pass
- ☐ Execute the tests and see success


---

©2016 Chef Software Inc. 3-26 

With the test defined it is now time to execute the tests and see the failure.


## Slide 27

## Move into the Cookbook Directory



A terminal window icon is shown on the left. The terminal displays the command `> cd httpd`. The rest of the terminal window is black, indicating no output or a continuation of the command.

©2016 Chef Software Inc. 3-27



To execute our tests using the tool Test Kitchen we need to be within the directory of the cookbook.

## Review the Existing Kitchen Configuration



```
> cat .kitchen.yml
```

```
---
driver:
  name: vagrant

provisioner:
  name: chef_zero

# Uncomment the following verifier to leverage Inspec instead of
# default verifier)
# verifier:
#   name: inspec
```

Before we employ Test Kitchen to execute the tests we need make changes to the existing Test Kitchen configuration file. The cookbook was automatically generated with a '.kitchen.yml'.

## Slide 29

## The Kitchen Driver

```
---
driver:
  name: vagrant

provisioner:
  name: chef_zero

platforms:
  - name: ubuntu-12.04
  - name: centos-6.5
...
```

The driver is responsible for creating a machine that we'll use to test our cookbook.

Example Drivers:

- docker
- vagrant

The first key is driver, which has a single key-value pair that specifies the name of the driver Kitchen will use when executed.

The driver is responsible for creating the instance that we will use to test our cookbook. There are lots of different drivers available--two very popular ones are the docker and vagrant driver.

Instructor Note: Testing on this remote workstation requires that we use Docker because Vagrant does not work within a virtual environment. Vagrant is the standard choice when working on your local workstation.

## The Kitchen Driver

```
---  
driver:  
  name: vagrant  
  
provisioner:  
  name: chef_zero  
  
platforms:  
  - name: ubuntu-12.04  
  - name: centos-6.5  
  
...
```

This tells Test Kitchen how to run Chef, to apply the code in our cookbook to the machine under test.

The default and simplest approach is to use chef\_zero.

The second key is provisioner, which also has a single key-value pair which is the name of the provisioner Kitchen will use when executed. This provisioner is responsible for how it applies code to the instance that the driver created. Here the default value is chef\_zero.

## Slide 31

## The Kitchen Driver

```
---  
driver:  
  name: vagrant  
  
provisioner:  
  name: chef_zero  
  
platforms:  
  - name: ubuntu-12.04  
  - name: centos-6.5  
  
...
```

This is a list of platforms on which we want to apply our recipes.

The third key is platforms, which contains a list of all the platforms that Kitchen will test against when executed. This should be a list of all the platforms that you want your cookbook to support.

## Slide 32

## The Kitchen Driver

```
platforms:
  - name: ubuntu-12.04
  - name: centos-6.5

suites:
  - name: default
    run_list:
      - recipe[httpd::default]
    attributes:
```

This section defines what we want to test. It includes the Chef run-list of recipes that we want to test.

We define a single suite named "default".

The fourth key is `suites`, which contains a list of all the test suites that Kitchen will test against when executed. Each suite usually defines a unique combination of run lists that exercise all the recipes within a cookbook.

In this example, this suite is named 'default'.



## Slide 33

## The Kitchen Driver

```
platforms:
  - name: ubuntu-12.04
  - name: centos-6.5

suites:
  - name: default
    run_list:
      - recipe[httpd::default]
    attributes:
```

The suite named "default" defines a run\_list.

Run the "workstation" cookbook's "default" recipe file.

This default suite will execute the run list containing: The httpd cookbook's default recipe.

## Remove Settings from the Kitchen Configuration

```
~/httpd/.kitchen.yml

---
driver:
  name: vagrant

provisioner:
  name: chef_zero

platforms:
  - name: ubuntu-14.04
  - name: centos-7.1

suites:
# ... REMAINDER OF THE KITCHEN CONFIGURATION FILE ...
```

The initial Test Kitchen configuration is set up in way for local development on non-virtual machine. Because we are currently on a virtual machine we cannot use vagrant. We are also not interested in those following platforms.

## Add Settings to the Kitchen Configuration

```
~/httpd/.kitchen.yml

---
driver:
  name: docker

provisioner:
  name: chef_zero

platforms:
  - name: centos-6.7


suites:
# ... REMAINDER OF THE KITCHEN CONFIGURATION FILE ...
```

There are many different drivers that Test Kitchen supports. The docker driver is configured to work on this virtual machine. At this moment we are only interested in verifying that the cookbook we develop works on this current platform.

Later we will return to this configuration file and add an additional platform.

## Slide 36

# CONCEPT




## Kitchen List

Kitchen defines a list of instances, or test matrix, based on the **platforms** multiplied by the **suites**.

PLATFORMS x SUITES

Running **kitchen list** will show that matrix.

---

©2016 Chef Software Inc. 3-36 

It is important to recognize that within the `.kitchen.yml` file we defined two fields that create a test matrix; the number of platforms we want to support multiplied by the number of test suites that we defined.

## Slide 37

## View the Test Matrix for Test Kitchen



```
> kitchen list
```

Instance	Driver	Provisioner	Verifier	Transport	Last Action
default-centos-67	Docker	ChefZero	Busser	Ssh	<Not Created>

©2016 Chef Software Inc. 3-37




We can visualize this test matrix by running the command ``kitchen list``.

In the output you can see that an instance is created in the list for every test suite and every platform. In our current file we have one suite, named 'default' and one platform CentOS.

Run the following command to verify that the Test Kitchen configuration file had been set up correctly.

## Slide 38

# CONCEPT




## Kitchen Create

```
$ kitchen create [INSTANCE|REGEXP|all]
```

Create one or more instances.

Create CentOS Instance

©2016 Chef Software Inc. 3-38 

Create or turn on a virtual or cloud instance for the platforms specified in the kitchen configuration.

Running 'kitchen create default-centos-67' would create the the one instance that uses the test suite on the platform we want.

Typing in that name would be tiring if you had a lot of instances. A shortcut can be used to target the same system 'kitchen create default' or 'kitchen create centos' or even 'kitchen create 67'. This is an example of using the Regular Expression (REGEXP) to specify an instance.

When you want to target all of the instances you can run 'kitchen create' without any parameters. This will create all instances. Seeing as how there is only one instance this will work well.

In our case, this command would use the Docker driver to create a docker image based on centos-6.7.

## Slide 39

# CONCEPT



## Kitchen Converge

```
$ kitchen converge [INSTANCE|REGEXP|all]
```

Create the instance (if necessary) and then apply the run list to one or more instances.

Create CentOS Instance

→

Install Chef

→

Apply the Run List

©2016 Chef Software Inc. 3-39 

Creating an image gives us a instance to test our cookbooks but it still would leave us with the work of installing chef and applying the cookbook defined in our .kitchen.yml run list.


So let's introduce you to the second kitchen command: 'kitchen converge'.

Converging an instance will create the instance if it has not already been created. Then it will install chef and apply that cookbook to that instance.

In our case, this command would take our image and install chef and apply the httpd cookbook's default recipe.

## Slide 40


# CONCEPT




## Kitchen Verify

```
$ kitchen verify [INSTANCE|REGEXP|all]
```

Create, converge, and verify one or more instances.



```
graph LR; A[Create CentOS Instance] --> B[Install Chef]; B --> C[Apply the Run List]; C --> D[Execute Tests]
```

©2016 Chef Software Inc. 3-40 

To verify an instance means to:

- Create a virtual or cloud instances, if needed
- Converge the instance, if needed
- And then execute a collection of defined tests against the instance



## Create the Virtual Instance



```
> kitchen create
```

```
-----> Starting Kitchen (v1.4.2)
-----> Creating <default-centos-67>...
    Sending build context to Docker daemon 26.11 kB
    Sending build context to Docker daemon
    Step 0 : FROM centos:centos6
    centos6: Pulling from centos
    47d44cb6f252: Pulling fs layer
    ...
    Finished creating <default-centos-67> (2m28.65s).
-----> Kitchen is finished. (2m29.39s)
```

Create the instance with the following command. Here Test Kitchen will ask the driver specified in the kitchen configuration file to provision an instance for us.

## Inspect the Virtual Instance



```
> kitchen login
```

```
$$$$$$ Running legacy login for 'Docker' Driver
```

```
Last login: Thu Feb 18 21:21:39 2016 from 172.17.42.1
```

```
[kitchen@4eae2dd9e741 ~]$
```

You can gain access to this virtual instance that we have created through the specified command. The login subcommand allows you to specify a parameter, which is the name of the instance that you want to log into. In your case, you only have one instance so Test Kitchen assumes you want to log into that one.

You are in now logged into a virtual instance on a virtual instance.

## Slide 43

## Exit the Virtual Instance



```
[kitchen@4eae2dd9e741 ~]$ exit
```

```
logout
```

```
Connection to localhost closed.
```

```
[chef@ip-172-31-14-170 httpd]$
```

Logging in to the virtual instance is useful to explore the platform or assist with troubleshooting your recipes they fail in perplexing ways. Right now, we are interested in executing the tests so logout of the instance with the 'exit' command and we will return to the workstation.

## Converge the Virtual Instance



```
> kitchen converge
```

```
----> Starting Kitchen (v1.4.2)
-----> Converging <default-centos-67>...
$$$$$$ Running legacy converge for 'Docker' Driver
...
-----> Installing Chef Omnibus (install only if missing)
Downloading https://www.chef.io/chef/install.sh to file...
resolving cookbooks for run list: ["httpd::default"]
...
Finished converging <default-centos-67> (0m27.64s).
-----> Kitchen is finished. (0m28.58s)
```

Creating the instance allows us to view the operating system but Chef is not installed and the cookbook recipe, defined in the run list of the default test suite, has not been applied to the system. To do that you need to run 'kitchen converge'. Converge will take care of all of that.

In this instance the default recipe of the httpd cookbook contains no resources. You have not written a single resource that defines your desired state. Before we do that we want to ensure the instance is not already in a state that perhaps already meets the expectations that we defined.

Slide 45

## Execute the Tests Against the Virtual Instance



```
> kitchen verify
```

```
-----> Starting Kitchen (v1.4.2)
-----> Setting up <default-centos-67>...
-----> Installing Busser (busser)
-----> Verifying <default-centos-67>...
-----> Running serverspec test suite
-----> Installing Serverspec..
-----> serverspec installed (version 2.24.1)
      /opt/chef/embedded/bin/ruby -
I/tmp/verifier/suites/serverspec -I/tmp/verifier/gems/gems/rspec-
support-3.3.0/lib:/tmp/verifier/gems/gems/rspec-core-3.3.2/lib
/opt/chef/embedded/bin/rspec --pattern
```

To verify the state of the instance with specification that we defined we use the 'kitchen verify' command. This command will install all the necessary testing tools, configure them, and then execute the test suite, and return to us the results.

Something that is important to mention is that we could have simply run this command from the start. When no previous instance exists, no instance has been created or converged, this command will automatically perform those two steps. When the instance is running, however, the verification step is only run.

## Slide 46

## Understanding the Failure Message

```
>>>>> Verify failed on instance <default-centos-67>.
>>>>> Please see .kitchen/logs/default-centos-67.log for more details
>>>>> -----Exception-----
>>>>> Class: Kitchen::ActionFailed
>>>>> Message: SSH exited (1) for command: [sh -c '

BUSSER_ROOT="/tmp/verifier"; export BUSSER_ROOT
GEM_HOME="/tmp/verifier/gems"; export GEM_HOME
GEM_PATH="/tmp/verifier/gems"; export GEM_PATH
GEM_CACHE="/tmp/verifier/gems/cache"; export GEM_CACHE

sudo -E /tmp/verifier/bin/busser test
']
>>>>> -----
[chef@ip-172-31-14-170 httpd]$
```

kitchen failure

command executed

Now, let's read the results from the kitchen verification to ensure that our expectations failed to be met.

When the command completes you will see a block of code that tells you that the verification failed and that an exception has occurred. Immediately your eyes will start to scan this block of text for some information about the failure and unfortunately you will not see anything to help you understand what is happening. Because what you are looking at is the test command executed by the Busser on the test instance but not the results of the command. To see those results you will need to scroll back up in your history.

## Examine the Test Kitchen Results

```
-----> serverspec installed (version 2.24.1)
/opt/chef/embedded/bin/ruby -I/tmp/verifier/suites/serverspec
-I/tmp/verifier/gems/gems/rspec-support-3.3.0/lib:/tmp/.../rspec-
core-3.3.2/lib /opt/chef/embedded/bin/rspec --pattern
/tmp/verifier/suites/serverspec/\*\*/\*_spec.rb --color --format
documentation --default-path /tmp/verifier/suites/serverspec
```

ServerSpec is the default verifier for Test Kitchen. The Busser tool installs it, configures it, and the executes it for you on the test instance.

Scroll back until you can find the message that tells you that serverspec is installed.

## Examine the Test Kitchen Results

```
-----> serverspec installed (version 2.24.1)
/opt/chef/embedded/bin/ruby -I/tmp/verifier/suites/serverspec
-I/tmp/verifier/gems/gems/rspec-support-3.3.0/lib:/tmp/.../rspec-
core-3.3.2/lib /opt/chef/embedded/bin/rspec --pattern
/tmp/verifier/suites/serverspec/\*\*/\*_spec.rb --color --format
documentation --default-path /tmp/verifier/suites/serverspec
```

ruby executable

include ruby libraries

rspec executable

rspec parameters

The line that immediately follows it is the command that is executed on the system. The ServerSpec verifier is running ruby, loading up the test suite libraries, executing the command rspec. The rspec command is being provided a number of command-line parameters that tell it: where to find the test files and what they look like; to colorize the output; and how to output the results.



## Examine the RSpec Results

```
httpd::default
  Command "curl http://localhost"
    stdout
      should match /Welcome Home/ (FAILED -1)
```

Failure Number

RSpec displays a summary of the results in the 'documentation' format. This format allows us to read the example groups and see that:

When running the specified command the standard out failed to match the value 'Welcome Home' anywhere within the results.

The results are displayed in color and in RSpec's documentation format. This shows us the example that we wrote in a hierarchal view. The indentation is intentional to show the nested relationships of the example groups and the example. The expectation in the example you defined failed, as we expected. The text will be displayed in red and provide a failure number. Details about the failure will be displayed below.

## Examine Failure #1

```
1) httpd::default Command "curl http://localhost" stdout should match ...
Failure/Error: its(:stdout) { should match(/Welcome Home/) }
  expected "" to match /Welcome Home/
  Diff:
  @@ -1,2 +1,2 @@
  -/Welcome Home/
  +" "
  /bin/sh -c curl\ http://localhost
# /.../serverspec/default_spec.rb:7:in `block (3 levels) in ...
```

actual results

difference

system command

spec file : line number

Each failure is displayed with their failure number, in order, in more detail in a failures section. A failure contains a number of details about the failure.

First it will display a sentence created out of the example groups and example that we defined. Below that it will display all the details about the failure that include: the actual results that were received; the difference between the actual and the expected results; the command run against the virtual instance; and the spec file and line number within that spec file where the failing expectation can be found.

## Examine the Test Summary


```
Finished in 0.20256 seconds (files took 0.60564 seconds to load)  
1 example, 1 failure
```

A final summary contains the length of execution time with the results shows that RSpec verified 1 example and found 1 failure.

After all the failures a final summary of the results will be displayed which shows us that our test suite contains 1 example and that 1 example failed to meet expectations.

## Slide 52

# EXERCISE




## Build a Reliable Cookbook

*This time it will be different.*

**Objective:**

- ✓ Create a cookbook
- ✓ Write tests that verifies the cookbook does what we want it to do
- ✓ Execute the tests and see failure
- ☐ Write the recipe to make the test pass
- ☐ Execute the tests and see success

---

©2016 Chef Software Inc. 3-52 

Now we know for certain that the test instance is not in our desired state. When we write the resources now in the default recipe to bring the instance to the desired state we can be certain that we have done it in a way that meets the expectations that we have established.

## Write the Default Recipe for the Cookbook

```
~/httpd/recipes/default.rb

#
# Cookbook Name:: httpd
# Recipe:: default
#
# Copyright (c) 2015 The Authors, All Rights Reserved.
package 'httpd'

file '/var/www/html/index.html' do
  content '<h1>Welcome Home!</h1>'
end

service 'httpd' do
  action [:enable, :start]
end
```

The following recipe defines three resources. These three resources express the desired state of an apache server that will serve up a simple page that contains the text 'Welcome Home!'.

The package will install all the necessary software on the operating system. The file will create an HTML file with the desired content at a location pre-defined by the web server. The service resource will start the web server and then ensure that if we reboot the system the web server will start up.

## Re-Converge the Virtual Instance



```
> kitchen converge
```

```
-----> Starting Kitchen (v1.4.2)
```

```
Converging 2 resources
```

```
Recipe: httpd::default
```

```
  * package[httpd] action install
```

```
    - install version 2.2.15-47.el6.centos of package httpd
```

```
  * file[/var/www/html/index.html] action create
```

```
    - ...
```

```
  * service[httpd] action enable
```

```
    - enable service service[httpd]
```

```
  * service[httpd] action start
```


```
    - start service service[httpd]
```

Whenever you make a change to the recipe it is important to run 'kitchen converge'. This command will apply the updated recipe to the state of the virtual instance.

In the output, you should see the resources that you defined being applied to the instance. The package, the file, and the actions of the service.

## Slide 55

# EXERCISE




## Build a Reliable Cookbook

*This time it will be different.*

**Objective:**

- ✓ Create a cookbook
- ✓ Write tests that verifies the cookbook does what we want it to do
- ✓ Execute the tests and see failure
- ✓ Write the recipe to make the test pass
- ☐ Execute the tests and see success

---

©2016 Chef Software Inc. 3-55 

Now with the desired state expressed in the default recipe and applied to the virtual instance it is time to see if the test we wrote initially will now pass. If it does, that means we got everything right in the configuration we wrote in the recipe. We can declare victory!

## Re-Verify the Virtual Instance



```
> kitchen verify
```

```
httpd::default
```

```
  Command "curl http://localhost"
```

```
    stdout
```

```
      should match /Welcome Home/
```


To verify the state of the virtual instance you run the 'kitchen verify' command. In the summary you should find the failing expectation no longer fails.

If it does fail, it is time to review the code you wrote in the recipe file and the spec file. When it was failing did you get a different failure than the one that we walked through? That probably means there is an error in the spec file. Did the test instance actually converge successfully? Sometimes output will scroll by and we don't have time to read it. I get it. Scroll back up and see if there was an error message tucked into the 'kitchen converge' you ran.



## Slide 57

# EXERCISE




## Build a Reliable Cookbook

*This time it will be different.*

**Objective:**

- ✓ Create a cookbook
- ✓ Write tests that verifies the cookbook does what we want it to do
- ✓ Execute the tests and see failure
- ✓ Write the recipe to make the test pass
- ✓ Execute the tests and see success

---


©2016 Chef Software Inc. 3-57 

So you've done it. You have done Test Driven Development (TDD). Wrote a test. Saw it fail. Wrote a unit of code. Saw it pass.

You created a cookbook. Wrote an expectation in the spec file. Saw the test fail. Wrote a recipe. Applied the recipe. Ran the tests and saw them pass.

## Slide 58

# DISCUSSION




## Discussion

What value is there is writing the tests before writing the recipes?

Why is it hard to write the tests before you write the recipe?

---

©2016 Chef Software Inc. 3-58 

Now that you participated in writing a test and then the recipe let's have a discussion.

Instructor Note: With large groups I often find it better to have individuals turn to the individuals around them, form groups of whatever size they feel comfortable, and have them take turns asking and answering the questions. When all the groups are done I then open the discussion up to the entire group allowing each group or individuals to share their answers.

Slide 59

# DISCUSSION



## Q&A

What questions can we answer for you?


©2016 Chef Software Inc.

3-59



Before we complete this section, let us pause for questions.

Slide 60

Morning	Afternoon
Introduction	Faster Feedback with Unit Testing
Why Write Tests? Why is that Hard?	Testing Resources in Recipes
Writing a Test First	Refactoring to Attributes
<b>Refactoring Cookbooks with Tests</b>	Refactoring to Multiple Platforms
©2016 Chef Software Inc.	3-60
	

You have performed almost all of the steps of TDD. Next we are going to use the tests to help us refactor the recipe we wrote. In a series of group exercises we will explore some of the important nuances of Test Kitchen's subcommands: converge and verify. And explore another subcommand named: test.

Slide 61

